On Computable Tree Functions (Algorithms in Algebraic Systems and Computation Theory)

AUTHOR(S):

Kimoto, Masahiro; Takahashi, Masako

数理解析研究所講究録 1268 巻 2002 年 138-150

On Computable Tree Functions*

Masahiro Kimoto† and Masako Takahashi‡

April 19, 2002

Abstract

In order to investigate the structure of computable functions over (binary) trees, we define two classes of recursive tree functions by extending the notion of recursive functions over natural numbers in two different ways, and also define the class of functions computable by while-programs over trees. Then we show that those classes coincide with the class of conjugates of recursive functions over natural numbers via a standard coding function (between trees and natural numbers). We also study how the choice of the coding function affects the result, and prove that the class of conjugates of recursive numeric functions via coding function φ remains same as above, so long as the conjugates of constructors suc and cons remain to be recursive.

0. Introduction

We consider in this paper a naive question: What are computable functions over trees. A simple answer for the question might be the following: a tree function is computable if and only if it is obtained from a computable function f over natural numbers as the conjugate $\varphi^{-1} \circ f \circ \varphi$ where φ is an appropriate coding function from trees to natural numbers and φ^{-1} is the decoding function.

Then a natural question arises: What coding functions are appropriate? Should not they be 'computable' in some sense? But then in what sense? Do we have to know the notion of computability of coding functions (from trees to natural numbers), before we study the notion of computability of tree functions?

Another question related to the tentative answer is: Does the notion of computability of tree functions depend on the choice of the coding function or not?

We try to answer the first question in a number of different ways, and also partially answer the other questions mentioned above.

Our motivation comes from the observation that a number of tree manipulating operations are found to be useful in well-known algorithms for sorting, searching, etc., and also in various application areas such as natural language processing. These are concrete examples of interesting 'computable' tree operations (or partial functions over trees), and certainly there should be many more. We would like to develop a structural theory of computable tree operations, which hopefully will fill the gap between conventional theory of computation and algorithmic aspects of trees.

In this paper, among various tree structures with or without labels, we concentrate on binary trees without labels. This is because binary trees have the important property that any tree structure (and moreover any finite sequence of tree structures) can be nicely represented by binary trees without any coding function involved. (See, e.g., [4].) Moreover, we believe that studying unlabelled binary trees,

---

*This paper is a revised version of the following paper. M. Kimoto and M. Takahashi: On computable tree functions, *Lecture Notes in Computer Science*, Vol. 1961, pp. 273 - 289. Springer-Verlag (2000). The major revision is to include some new results (Lemmas 4.4 and 4.5, and Theorem 4.6) and to make the proof of a previous result (Corollary 4.7) much shorter.

†木本正裕 コンパックコンピュータ株式会社 Compaq Computer K.K., Tokyo 167-8533 Japan. E-mail: Masahiro.Kimoto@jp.compaq.com

‡高橋正子 国際基督教大学 Department of Information Science, International Christian University, Tokyo 181-8585 Japan. E-mail: mth@icu.ac.jp

a degenerate case of labelled binary trees, would be essential for studying the general case.

This paper is organized as follows. In the rest of this section, we summarize some notations and known facts about computable functions over $\mathbf{N} = \{0, 1, 2, ...\}$, the set of natural numbers. In Section 1, we present basic definitions about binary trees and their functions, and some of their basic properties. Then in Section 2 we introduce the notion of primitive recursive functions over binary trees, and show that the class of those functions is precisely that of conjugates of conventional primitive recursive functions (over $\mathbf{N}$) via the standard coding function (between binary trees and natural numbers). In Section 3, two classes of recursive tree functions are introduced, one of which depends on the coding function and the other does not. Then we prove that the two classes are equal to the class of conjugates of conventional recursive functions (over $\mathbf{N}$), and also equal to the class of functions computable by while-programs over binary trees. In the last section, we study how the choice of coding function affects the results; indeed we give a necessary and sufficient condition for coding functions to yield the above results.

In this paper, a function from a subset of $\mathbf{N}^n$ to $\mathbf{N}$ is called a *function over* $\mathbf{N}$ or a *numeric function*, and for such a function f we write $f : \mathbf{N}^n \rightharpoonup \mathbf{N}$ (using a special symbol $\rightharpoonup$ rather than the usual $\rightarrow$). Note that in the literature these functions are called 'partial' functions and we may occasionally use that word to stress the point. As usual, the notation $f : X^n \rightarrow Y$ means that f is a (total) function from X to Y.

Recall that the class $\mathbf{PR}(\mathbf{N})$ of primitive recursive functions over $\mathbf{N}$ is defined as the least class that contains the successor function $\mathsf{suc} : \mathbf{N} \rightarrow \mathbf{N}$, the zero functions $\mathsf{zero}_n : \mathbf{N}^n \rightarrow \mathbf{N}$ and projection functions $p_{n,i} : \mathbf{N}^n \rightarrow \mathbf{N}$, and is closed under composition and primitive recursion. Also recall that the class $\mathbf{R}(\mathbf{N})$ of recursive (partial) functions over $\mathbf{N}$ is defined as the least class that contains primitive recursive functions over $\mathbf{N}$ and their minimization functions, and is closed under composition. The minimization function $\mu f : \mathbf{N}^n \rightharpoonup \mathbf{N}$ of $f : \mathbf{N}^{n+1} \rightharpoonup \mathbf{N}$ is defined by

$$(\mu f)(\vec{x}) = x \iff \forall y \leq x[f(\vec{x}, y) = 0 \Leftrightarrow y = x].$$

The class $\mathbf{R}(\mathbf{N})$ is known to be equal to the class of functions computed by a (high-level) while-*program* of the form;

$$\mathsf{input}(\vec{x}); S_1; S_2; ...; S_k; \mathsf{output}(y)$$

where each S_i is a *statement*, which is either an assignment statement or a while-statement. Here an *assignment statement* is of the form

$$x := f(\vec{y})$$

and a while-*statement* is of the form

$$\mathsf{while}\ f(\vec{y}) \neq 0\ \mathsf{do}\ [S'_1; S'_2; ...; S'_l]$$

where f is an arbitrary primitive recursive function (in $\mathbf{PR}(\mathbf{N})$), x, y are variables (to store elements of $\mathbf{N}$), $\vec{x}$ is a sequence of distinct (input) variables, and $\vec{y}$ is any sequence of variables. S'_j's in the while-statement are statements. The (partial) function $f_P : \mathbf{N}^n \rightharpoonup \mathbf{N}$ *computed by* a while-program P with n input variables is defined as usual, assuming that the initial values of all variables other than input variables are set to be 0. For more about the theory of computation and related subjects, see, e.g., [7].

In the literature, the notions of $\mathbf{PR}(\mathbf{N})$ and $\mathbf{R}(\mathbf{N})$ have been extended to functions over algebraic structures other than $\mathbf{N}$ (see, e.g., [1], [2], [8], [9]). The present work may be considered as an extention of studies on word functions in [2] and [8] to the case of functions over binary trees. As a related work, we also note the seminal work [5] by J. McCarthy, which has taken a completely different approach to studying computable functions over binary trees (with labelled leaves).

1. Basic Definitions

1.1 Definition The set $\mathbf{T}$ of *binary trees* (or simply *trees*) is defined recursively, as follows.

1. $\mathsf{nil} \in \mathbf{T}$.
2. If $t_1, t_2 \in \mathbf{T}$, then $\mathsf{cons}(t_1, t_2) \in \mathbf{T}$.

When $t = \mathsf{cons}(t_1, t_2)$, we write $\mathsf{left}(t) = t_1$ and $\mathsf{right}(t) = t_2$. For each $t \in \mathbf{T}$, we define the size $|t| (\in \mathbf{N})$ of t recursively as

$$|\mathsf{nil}| = 0, \qquad |\mathsf{cons}(t_1, t_2)| = |t_1| + |t_2| + 1.$$

It is well-known (see e.g. [4]) that for each $n \in \mathbf{N}$ the number of binary trees of the size n is equal to the Catalan number $B(n) = \frac{1}{n+1}\binom{2n}{n}$.

1.2 Definition For $s, t \in \mathbf{T}$, we write $s \prec t$ if one of the following conditions holds;

1. $|s| < |t|$.
2. $|s| = |t|$ and $\mathsf{left}(s) \prec \mathsf{left}(t)$.
3. $|s| = |t|$, $\mathsf{left}(s) = \mathsf{left}(t)$ and $\mathsf{right}(s) \prec \mathsf{right}(t)$.

It is not difficult to see that the reflexive closure $\preceq$ of $\prec$ is a total order in $\mathbf{T}$, and moreover the ordered set $(\mathbf{T}, \preceq)$ is isomorphic to $(\mathbf{N}, \leq)$ through bijection $\nu(t) = \mathrm{card}\{s \in \mathbf{T} | s \prec t\}$. (See [3].)

1.3 Definition Let us write $\mathsf{next}(t)$ for $\min\{s \in \mathbf{T} | t \prec s\}$, where min refers to the order $\preceq$. Note that the inverse $\tau : \mathbf{N} \to \mathbf{T}$ of our standard coding function $\nu : \mathbf{T} \to \mathbf{N}$ can be described by using $\mathsf{next} : \mathbf{T} \to \mathbf{T}$ as

$$\tau(n) = \nu^{-1}(n) = \mathsf{next}^n(\mathsf{nil}).$$

1.4 Definition For each $n \in \mathbf{N}$, let us write $\underline{n}$ ($\overline{n}$, respectively) for the minimum (maximum) binary tree of the size n; that is,

$$\underline{0} = \mathsf{nil}, \qquad \underline{n+1} = \mathsf{cons}(\mathsf{nil}, \underline{n}),$$

$$\overline{0} = \mathsf{nil}, \qquad \overline{n+1} = \mathsf{cons}(\overline{n}, \mathsf{nil}).$$

We also write $\underline{\mathbf{N}} = \{\underline{n} | n \in \mathbf{N}\}$ and $\overline{\mathbf{N}} = \{\overline{n} | n \in \mathbf{N}\}$.

Using these notations, we can give recursive descriptions of functions $\mathsf{next} : \mathbf{T} \to \mathbf{T}$ and $\nu : \mathbf{T} \to \mathbf{N}$. For the proofs, see [3].

1.5 Lemma

- If $t \in \overline{\mathbf{N}}$, then $\mathsf{next}(t) = \underline{|t| + 1}$.
- If $t = \mathsf{cons}(t', t'') \notin \overline{\mathbf{N}}$ and
  - if $t'' \notin \overline{\mathbf{N}}$, then $\mathsf{next}(t) = \mathsf{cons}(t', \mathsf{next}(t''))$,
  - if $t'' \in \overline{\mathbf{N}}$ and $t' \notin \overline{\mathbf{N}}$, then $\mathsf{next}(t) = \mathsf{cons}(\mathsf{next}(t'), \underline{|t''|})$,
  - If $t'' \in \overline{\mathbf{N}}$ and $t' \in \overline{\mathbf{N}}$, then $\mathsf{next}(t) = \mathsf{cons}(\underline{|t'| + 1}, \underline{|t''| - 1})$.

  Note that in the last subcase we have $|t''| > 0$ since $t' \in \overline{\mathbf{N}}$ and $t \notin \overline{\mathbf{N}}$.

1.6 Lemma

- If $t = \mathsf{nil}$, then $\nu(t) = 0$.
- If $t = \mathsf{cons}(t', t'')$, then

$$\begin{aligned}\nu(t) \;=\; & \textstyle\sum_{n<|t|} B(n) \\ & + \textstyle\sum_{n<|t'|}(B(n) \times B(|t| - n - 1)) \\ & + (\nu(t') - \nu(|t'|)) \times B(|t''|) \\ & + (\nu(t'') - \nu(\underline{|t''|})).\end{aligned}$$

The notion of conjugates, which plays an essential role in this paper, can be defined, as follows.

1.7 Definition Given a bijection $\varphi : X \to Y$ and a function $f : Y^n \rightharpoonup Y$ we define the *conjugate* $f_\varphi : X^n \rightharpoonup X$ of f via φ by

$$f_\varphi(x_1, x_2, ..., x_n) = x \iff f(\varphi(x_1), \varphi(x_2), ..., \varphi(x_n)) = \varphi(x).$$

For simplicity, we may write $\varphi^{-1} \circ f \circ \varphi$ for the conjugate f_φ even if f is not unary. Note that the conjugate of f_φ via φ^{-1} is f. For example, $\mathsf{next} : \mathbf{T} \to \mathbf{T}$ is the conjugate suc_ν of $\mathsf{suc} : \mathbf{N} \to \mathbf{N}$ via $\nu : \mathbf{T} \to \mathbf{N}$, since $\mathsf{next}(\nu^{-1}(n)) = \mathsf{next}^{n+1}(\mathsf{nil}) = \nu^{-1}(n+1) = \nu^{-1}(\mathsf{suc}(n))$, thus $\mathsf{next} = \nu^{-1} \circ \mathsf{suc} \circ \nu = \mathsf{suc}_\nu$. Also, $\mathsf{suc} = \mathsf{next}_\tau$, the conjugate of next via $\tau = \nu^{-1}$.

2. Primitive Recursive Functions over T

In this section, we define the notion of primitive recursive functions over $\mathbf{T}$, and compare them with conjugates of primitive recursive functions over $\mathbf{N}$.

2.1 Definition The class $\mathbf{PR}(\mathbf{T})$ of *primitive recursive functions over* $\mathbf{T}$ is defined recursively, as follows.

1. (constructors) The binary function $\mathsf{cons} : \mathbf{T}^2 \to \mathbf{T}$ and n-ary constant functions $\mathsf{nil}_n : \mathbf{T}^n \to \mathbf{T}$ such that $\mathsf{nil}_n(\vec{t}) = \mathsf{nil}$ $(n \geq 0)$ belong to $\mathbf{PR}(\mathbf{T})$.
2. (projections) $p_{n,i} : \mathbf{T}^n \to \mathbf{T}$ such that $p_{n,i}(t_1, t_2, ..., t_n) = t_i$ belong to $\mathbf{PR}(\mathbf{T})$ $(1 \leq i \leq n)$.
3. (composition) $\mathbf{PR}(\mathbf{T})$ is closed under composition. That is, if $g : \mathbf{T}^l \to \mathbf{T}$ and $g_1, g_2, ..., g_l : \mathbf{T}^n \to \mathbf{T}$ belong to $\mathbf{PR}(\mathbf{T})$, then so does the function $f : \mathbf{T}^n \to \mathbf{T}$ defined by $f(\vec{t}) = g(g_1(\vec{t}), ..., g_l(\vec{t}))$.
4. (primitive recursion) If $g : \mathbf{T}^n \to \mathbf{T}$ and $h : \mathbf{T}^{n+4} \to \mathbf{T}$ belong to $\mathbf{PR}(\mathbf{T})$, then so does the function $f : \mathbf{T}^{n+1} \to \mathbf{T}$ defined by

$$\begin{cases} f(\vec{t}, \mathsf{nil}) = g(\vec{t}), \\ f(\vec{t}, \mathsf{cons}(t', t'')) = h(\vec{t}, t', t'', f(\vec{t}, t'), f(\vec{t}, t'')). \end{cases}$$

We will denote the function f in 3 by $g \circ (g_1, ..., g_l)$, and the one in 4 by $h * g$.

2.2 Examples The following functions belong to $\mathbf{PR}(\mathbf{T})$.

1. The unary functions $\mathsf{left} : \mathbf{T} \to \mathbf{T}$ and $\mathsf{right} : \mathbf{T} \to \mathbf{T}$ defined by

$$\begin{cases} \mathsf{left}(\mathsf{nil}) = \mathsf{nil}, \\ \mathsf{left}(\mathsf{cons}(t', t'')) = t', \end{cases}$$

$$\begin{cases} \mathsf{right}(\mathsf{nil}) = \mathsf{nil}, \\ \mathsf{right}(\mathsf{cons}(t', t'')) = t''. \end{cases}$$

2. The minimum tree function $\mathsf{mnt} : \mathbf{T} \to \mathbf{T}$ to assign the minimum tree of the same size as the argument (i.e., $\mathsf{mnt}(t) = \underline{|t|}$) can be defined by primitive recursion:

$$\begin{cases} \mathsf{mnt}(\mathsf{nil}) = \mathsf{nil}, \\ \mathsf{mnt}(\mathsf{cons}(t', t'')) = \mathsf{cons}(\mathsf{nil}, \mathsf{gr}(\mathsf{mnt}(t'), \mathsf{mnt}(t''))) \end{cases}$$

where $\mathsf{gr} : \mathbf{T}^2 \to \mathbf{T}$ is the function to graft the first argument at the rightmost leaf of the second argument;

$$\mathsf{gr}(t, \mathsf{nil}) = t, \qquad \mathsf{gr}(t, \mathsf{cons}(t', t'')) = \mathsf{cons}(t', \mathsf{gr}(t, t'')).$$

Likewise, we can define the maximum tree function $\mathsf{mxt} : \mathbf{T} \to \mathbf{T}$ such that $\mathsf{mxt}(t) = \overline{|t|}$ by

$$\begin{cases} \mathsf{mxt}(\mathsf{nil}) = \mathsf{nil}, \\ \mathsf{mxt}(\mathsf{cons}(t', t'')) = \mathsf{cons}(\mathsf{gl}(\mathsf{mxt}(t'), \mathsf{mxt}(t'')), \mathsf{nil}) \end{cases}$$

where

$$\mathsf{gl}(t, \mathsf{nil}) = t, \qquad \mathsf{gl}(t, \mathsf{cons}(t', t'')) = \mathsf{cons}(\mathsf{gl}(t, t'), t'').$$

3. The function $\mathsf{nil?} : \mathbf{T} \to \mathbf{T}$ to tell whether the given tree is nil or not can be defined by

$$\begin{cases} \mathsf{nil?}(\mathsf{nil}) = \mathsf{true}, \\ \mathsf{nil?}(\mathsf{cons}(t', t'')) = \mathsf{false}. \end{cases}$$

Here we define true and false by nil and cons(nil, nil), respectively. The characteristic functions $\underline{\mathbf{N}}? : \mathbf{T} \to \mathbf{T}$ ($\overline{\mathbf{N}}? : \mathbf{T} \to \mathbf{T}$, respectively) of the sets $\underline{\mathbf{N}} = \{\underline{n} | n \in \mathbf{N}\}$ ($\overline{\mathbf{N}} = \{\overline{n} | n \in \mathbf{N}\}$) are defined by

$$\begin{cases} \underline{\mathbf{N}}?(\mathsf{nil}) = \mathsf{true}, \\ \underline{\mathbf{N}}?(\mathsf{cons}(t', t'')) = \mathsf{if}(\mathsf{nil?}(t'), \underline{\mathbf{N}}?(t''), \mathsf{false}), \end{cases}$$

$$\begin{cases} \overline{\mathbf{N}}?(\mathsf{nil}) = \mathsf{true}, \\ \overline{\mathbf{N}}?(\mathsf{cons}(t', t'')) = \mathsf{if}(\mathsf{nil?}(t''), \overline{\mathbf{N}}?(t'), \mathsf{false}), \end{cases}$$

where

$$\begin{cases} \mathsf{if}(\mathsf{nil}, t, s) = t, \\ \mathsf{if}(\mathsf{cons}(t', t''), t, s) = s. \end{cases}$$

4. The function $\mathsf{next} : \mathbf{T} \to \mathbf{T}$ is primitive recursive, because

$$\begin{cases} \mathsf{next}(\mathsf{nil}) = \mathsf{cons}(\mathsf{nil}, \mathsf{nil}), \\ \mathsf{next}(\mathsf{cons}(t', t'')) = \mathsf{if}(\overline{\mathbf{N}}?(\mathsf{cons}(t', t'')), u, v) \end{cases}$$

where

$$u = \mathsf{cons}(\mathsf{nil}, \mathsf{mnt}(\mathsf{cons}(t', t''))) \quad (= \underline{|\mathsf{cons}(t', t'')| + 1}),$$

$$v = \mathsf{if}(\overline{\mathbf{N}}?(t''), \mathsf{if}(\overline{\mathbf{N}}?(t'), w_1, w_2), \mathsf{cons}(t', \mathsf{next}(t''))),$$

$$w_1 = \mathsf{cons}(\mathsf{cons}(\mathsf{nil}, \mathsf{mnt}(t')), \mathsf{right}(\mathsf{mnt}(t''))) \quad (= \mathsf{cons}(\underline{|t'| + 1}, \underline{|t''| - 1})),$$

$$w_2 = \mathsf{cons}(\mathsf{next}(t'), \mathsf{mnt}(t'')).$$

In what follows, in studying the relation between conjugates of primitive recursive numeric functions and primitive recursive tree functions, we find it is useful to have a reasonable embedding of the class $\mathbf{PR(N)}$ of primitive recursive functions over $\mathbf{N}$ into the class $\mathbf{PR(T)}$ of primitive recursive functions over $\mathbf{T}$.

2.3 Lemma For each n-ary numeric function $f \in \mathbf{PR}(\mathbf{N})$, there exists an n-ary tree function $\underline{f} \in \mathbf{PR}(\mathbf{T})$ such that for each $m_1, m_2, ..., m_n \in \mathbf{N}$

$$\underline{f}(\underline{m_1}, \underline{m_2}, ..., \underline{m_n}) = \underline{f(m_1, m_2, ..., m_n)}.$$

Proof We define tree functions $\underline{f}$ recursively, as follows:

1. $\underline{\mathsf{zero}_n} = \mathsf{nil}_n$.
   $\underline{\mathsf{suc}(t)} = \mathsf{cons}(\mathsf{nil}, t)$.
   $\underline{p_{n,i} : \mathbf{N}^n \to \mathbf{N}} = p_{n,i} : \mathbf{T}^n \to \mathbf{T}$.

2. $\underline{g \circ (g_1, ..., g_l)} = \underline{g} \circ (\underline{g_1}, ..., \underline{g_l})$.

3. $\underline{h * g} = h' * \underline{g}$ where $h'(\vec{t}, t', t'', s', s'') = \underline{h}(\vec{t}, t'', s'')$.
   Here the notation $h * g$ in the lefthand side stands for the numeric function defined by primitive recursion from g and h; that is,

$$(h * g)(\vec{x}, 0) = g(\vec{x}), \quad (h * g)(\vec{x}, x+1) = h(\vec{x}, x, (h * g)(\vec{x}, x)).$$

Then it is easy to see by induction on $\mathbf{PR}(\mathbf{N})$ that the functions $\underline{f}$ so defined satisfy the required property. □

2.4 Lemma The conjugates $\mathsf{cons}_\tau : \mathbf{N}^2 \to \mathbf{N}$, and $\mathsf{left}_\tau, \mathsf{right}_\tau : \mathbf{N} \to \mathbf{N}$ via the standard decoding function $\tau = \nu^{-1} : \mathbf{N} \to \mathbf{T}$ are primitive recursive.

Proof By definition of cons_τ, we have

$$\mathsf{cons}_\tau(m_1, m_2) = \nu(\mathsf{cons}(\tau(m_1), \tau(m_2))) = \nu(\mathsf{cons}(t_1, t_2)) = \nu(t)$$

where $t_i = \tau(m_i)$ $(i = 1, 2)$ and $t = \mathsf{cons}(t_1, t_2)$. Then by applying Lemma 1.6 we get

$$\begin{aligned} \mathsf{cons}_\tau(m_1, m_2) \;=\; & \textstyle\sum_{n < f(m_1)+f(m_2)+1} B(n) \\ & + \textstyle\sum_{n < f(m_1)} (B(n) \times B(f(m_1) + f(m_2) \dot{-} n)) \\ & + (m_1 \dot{-} g(m_1)) \times B(f(m_2)) \\ & + (m_2 \dot{-} g(m_2)) \end{aligned}$$

where

$$B(m) = (2 \times m)! \div ((m+1)! \times m!),$$

$$f(m) = \mu x < m\,[m < \textstyle\sum_{n \le x} B(n)] \quad (= |\tau(m)|\,),$$

$$g(m) = \textstyle\sum_{n < f(m)} B(n) \quad (= \nu(\underline{|\tau(m)|}) = \nu(\mathsf{mnt}(\tau(m)))\,).$$

The functions left_τ and right_τ can be expressed as

$$\mathsf{left}_\tau(m) = \mu x < m\,[\exists y < m[\mathsf{cons}_\tau(x, y) = m]],$$

$$\mathsf{right}_\tau(m) = \mu y < m\,[\exists x < m[\mathsf{cons}_\tau(x, y) = m]].$$

Thus the three functions are primitive recursive. □

2.5 Lemma For each $t \in \mathbf{T}$, we write $\theta(t) = \underline{\nu(t)}$. Then the function $\theta : \mathbf{T} \to \mathbf{T}$ satisfies the following:

- θ is primitive recursive; that is, $\theta \in \mathbf{PR}(\mathbf{T})$.

- θ gives an isomorphism (i.e., order preserving bijection) between $(\mathbf{T}, \preceq)$ and $(\underline{\mathbf{N}}, \leq)$ where $\leq$ is the total order in $\underline{\mathbf{N}}$ defined by $\underline{n} \leq \underline{m} \Leftrightarrow n \leq m$.
- There exists $g : \mathbf{T} \to \mathbf{T}$ in $\mathbf{PR(T)}$ such that

$$\forall t \in \mathbf{T}, \forall n \in \mathbf{N}[\theta(t) = \underline{n} \iff t = g(\underline{n})].$$

By abuse of notation, we write θ^{-1} for the function g.

Proof

- $\theta : \mathbf{T} \to \mathbf{T}$ is primitive recursive, because

$$\begin{aligned} \theta(\mathsf{cons}(t', t'')) &= \underline{\nu(\mathsf{cons}(t', t''))} \\ &= \underline{\mathsf{cons}_\tau(\nu(t'), \nu(t''))} \\ &= \underline{\mathsf{cons}_\tau}(\underline{\nu(t')}, \underline{\nu(t'')}) \\ &= \underline{\mathsf{cons}_\tau}(\theta(t'), \theta(t'')). \end{aligned}$$

(For the definition of $\underline{\mathsf{cons}_\tau}$, see 2.3.)

- The function θ, being the compostion of two isomorphisms $\nu : (\mathbf{T}, \preceq) \to (\mathbf{N}, \leq)$ and $\_ : (\mathbf{N}, \leq) \to (\underline{\mathbf{N}}, \leq)$, gives an isomorphism between $(\mathbf{T}, \preceq)$ and $(\underline{\mathbf{N}}, \leq)$.
- Define $g : \mathbf{T} \to \mathbf{T}$ by primitive recursion;

$$\begin{cases} g(\mathsf{nil}) = \mathsf{nil}, \\ g(\mathsf{cons}(s, t)) = \mathsf{next}(g(t)). \end{cases}$$

Then $g(\underline{0}) = \mathsf{nil}$, and $g(\underline{n+1}) = g(\mathsf{cons}(\mathsf{nil}, \underline{n})) = \mathsf{next}(g(\underline{n}))$. Therefore by induction $g(\underline{n}) = \mathsf{next}^n(\mathsf{nil}) = \tau(n)$; thus

$$t = g(\underline{n}) = \tau(n) \iff \nu(t) = n \iff \theta(t) = \underline{n}. \qquad \square$$

2.6 Lemma If $f \in \mathbf{PR(N)}$, then $f_\nu = \theta^{-1} \circ \underline{f} \circ \theta$. That is,

$$f_\nu(t_1, ..., t_n) = \theta^{-1}(\underline{f}(\theta(t_1), ..., \theta(t_1))).$$

Proof By Lemma 2.3, we have

$$\begin{aligned} f_\nu(\vec{t}) = s &\iff f(\nu(\vec{t})) = \nu(s) \\ &\iff \underline{f}(\theta(\vec{t})) = \underline{f(\nu(\vec{t}))} = \underline{\nu(s)} = \theta(s) \\ &\iff \theta^{-1}(\underline{f}(\theta(\vec{t}))) = s. \end{aligned}$$

$\square$

2.7 Corollary If $f \in \mathbf{PR(N)}$, then $f_\nu \in \mathbf{PR(T)}$.

Proof Immediate from Lemmas 2.6, 2.5 and 2.3. $\square$

2.8 Theorem $\mathbf{PR(N)}_\nu = \mathbf{PR(T)}$, where $\mathbf{PR(N)}_\nu = \{f_\nu | f \in \mathbf{PR(N)}\}$.

Proof Since inclusion $\subseteq$ has been proved, we now verify

$$\forall g \in \mathbf{PR(T)}, \exists f \in \mathbf{PR(N)}\ [g = f_\nu]$$

by induction on $\mathbf{PR(T)}$.

1. $\mathsf{nil}_n = \nu_{-1} \circ \mathsf{zero}_n \circ \nu = (\mathsf{zero}_n)_\nu$.
   $\mathsf{cons} = \nu_{-1} \circ \nu \circ \mathsf{cons} \circ \nu_{-1} \circ \nu = (\mathsf{cons}_\tau)_\nu$ where $\mathsf{cons}_\tau \in \mathbf{PR(N)}$.
   $(p_{n,i} : \mathbf{T}^n \to \mathbf{T}) = \nu_{-1} \circ (p_{n,i} : \mathbf{N}^n \to \mathbf{N}) \circ \nu = (p_{n,i} : \mathbf{N}^n \to \mathbf{N})_\nu$.

2. If $g = g_0 \circ (g_1, ..., g_l)$ and $g_j = (f_j)_\nu$ $(j = 1, ...l)$, then

$$g = (f_0 \circ (f_1, ..., f_l))_\nu.$$

3. Let $g = g_1 * g_0$ and $g_j = (f_j)_\nu$ where $f_j \in \mathbf{PR(N)}$ $(j = 0, 1)$. It suffies to show that the function $f = g_\tau = \nu \circ g \circ \tau$ belongs to $\mathbf{PR(N)}$, since $g = f_\nu$. From the definition, we have

$$\begin{aligned}
f(\vec{m}, 0) &= (\nu \circ g \circ \tau)(\vec{m}, 0) \\
&= \nu(g(\tau(\vec{m}), \mathsf{nil})) \\
&= \nu(g_0(\tau(\vec{m}))) \\
&= f_0(\vec{m}), \\
f(\vec{m}, m+1) &= (\nu \circ g \circ \tau)(\vec{m}, m+1) \\
&= \nu(g(\tau(\vec{m}), \mathsf{cons}(t', t''))) \\
&\qquad \text{where we write } \mathsf{cons}(t', t'') = \tau(m+1) \\
&= \nu(g_1(\tau(\vec{m}), t', t'', g(\tau(\vec{m}), t'), g(\tau(\vec{m}), t''))) \\
&= f_1(\vec{m}, m', m'', f(\vec{m}, m'), f(\vec{m}, m''))
\end{aligned}$$

   where

$$m' = \nu(t') = \nu(\mathsf{left}(\tau(m+1))) = \mathsf{left}_\tau(m+1),$$
$$m'' = \nu(t'') = \nu(\mathsf{right}(\tau(m+1))) = \mathsf{right}_\tau(m+1).$$

   Since $\mathsf{left}_\tau(m+1), \mathsf{right}_\tau(m+1) < m+1$, and $\mathsf{left}_\tau, \mathsf{right}_\tau \in \mathbf{PR(N)}$, the description of f implies that f is primitive recursive. (cf. [6]) □

3. Recursive Functions over T

In this section, we define two classes of recursive functions over $\mathbf{T}$ and the class of functions computable by while programs over $\mathbf{T}$. Then we study their properties in connection with conjugates of recursive functions over $\mathbf{N}$. As in the case of numeric functions, we call a function from a subset of $\mathbf{T}^n$ to $\mathbf{T}$ a *function over* $\mathbf{T}$ or a *tree function.*

3.1 Definition We define the class $\mathbf{R(T)}$ of *recursive functions* over $\mathbf{T}$ recursively, as follows.

1. (primitive recursive functions) $\mathbf{PR(T)} \subseteq \mathbf{R(T)}$.

2. (minimization) If $f : \mathbf{T}^{n+1} \to \mathbf{T}$ is primitive recursive, then the (partial) function $\mu_{\mathbf{T}} f : \mathbf{T}^n \rightharpoonup \mathbf{T}$ defined by

$$(\mu_{\mathbf{T}} f)(\vec{t}) = t \iff \forall s \preceq t[f(\vec{t}, s) = \mathsf{nil} \Leftrightarrow s = t]$$

   belongs to $\mathbf{R(T)}$. We call $\mu_{\mathbf{T}} f : \mathbf{T}^n \rightharpoonup \mathbf{T}$ the $\mathbf{T}$-*minimization function* of f along the values $\tau(0), \tau(1), \tau(2), ...$ of $\tau : \mathbf{N} \to \mathbf{T}$.

3. (composition) $\mathbf{R(T)}$ is closed under composition. That is, if partial functions $g : \mathbf{T}^l \rightharpoonup \mathbf{T}$ and $g_1, g_2, ..., g_l : \mathbf{T}^n \rightharpoonup \mathbf{T}$ belong to $\mathbf{R(T)}$, then so does the partial function $f : \mathbf{T}^n \rightharpoonup \mathbf{T}$ defined by

$$f(\vec{t}) = s \iff \exists s_1, ..., s_l \in \mathbf{T}[g_1(\vec{t}) = s_1, ..., g_l(\vec{t}) = s_l, g(s_1, ..., s_l) = s].$$

   As before we will write $g \circ (g_1, ..., g_l)$ for the function f.

One may wonder whether Definition 3.1.2 is the only reasonable way of defining the minimization function. For example, how about the following as an alternative?

3.2 Definition For a primitive recursive function $f : \mathbf{T}^{n+1} \to \mathbf{T}$, define a partial function $\mu_{\underline{\mathbf{N}}} f : \mathbf{T}^n \rightharpoonup \mathbf{T}$ by

$$(\mu_{\underline{\mathbf{N}}} f)(\vec{t}) = t \iff \exists m \in \mathbf{N}[t = \underline{m} \wedge \forall n \leq m[f(\vec{t}, \underline{n}) = \mathsf{nil} \Leftrightarrow n = m]].$$

We call $\mu_{\underline{\mathbf{N}}} f : \mathbf{T}^n \rightharpoonup \mathbf{T}$ the $\underline{\mathbf{N}}$-*minimization* of $f : \mathbf{T}^{n+1} \rightharpoonup \mathbf{T}$ along the values $\underline{0}, \underline{1}, \underline{2}, \ldots$ of bijection $\underline{\ } : \mathbf{N} \to \underline{\mathbf{N}}$. We will write $\mathbf{R}_{\underline{\mathbf{N}}}(\mathbf{T})$ for the class of tree functions defined exactly as $\mathbf{R}(\mathbf{T})$ except that we replace the minimization operator $\mu_{\mathbf{T}}$ with $\mu_{\underline{\mathbf{N}}}$. The (partial) functions in $\mathbf{R}_{\underline{\mathbf{N}}}(\mathbf{T})$ are called $\underline{\mathbf{N}}$-*recursive functions.*

Next, we define the notion of (high-level) while-programs over $\mathbf{T}$.

3.3 Definition A while-*program* over $\mathbf{T}$ is of the form;

$$\mathsf{input}(\vec{x}); S_1; S_2; \ldots; S_k; \mathsf{output}(y)$$

where each S_i is a statement, which is either an assignment statement or a while-statement. An *assignment statement* is of the form

$$x := f(\vec{y})$$

and a while-*statement* is of the form

$$\mathsf{while}\ f(\vec{y}) \neq \mathsf{nil}\ \mathsf{do}\ [S'_1; S'_2; \ldots; S'_l].$$

Here, f is an arbitrary primitive recursive function (in $\mathbf{PR}(\mathbf{T})$), x, y are variables (to store trees in $\mathbf{T}$), $\vec{x}$ is a sequence of distinct (input) variables, and $\vec{y}$ is any sequence of variables. S'_j's in the while-statement are (either assignment or while-) statements.

The (partial) function *computed by* a while-program P over $\mathbf{T}$ is defined as usual, assuming that the initial values of all variables other than input variables are set to nil, and it is denoted by $f_P : \mathbf{T}^n \rightharpoonup \mathbf{T}$ where n is the number of input variables of the program P.

First, we study the relation between the three classes; the class of conjugates of recursive functions over $\mathbf{N}$, the class $\mathbf{R}_{\underline{\mathbf{N}}}(\mathbf{T})$ of $\underline{\mathbf{N}}$-recursive functions, and that of functions computable by while-programs over $\mathbf{T}$.

3.4 Lemma We extend the definition of $\underline{f}$ in lemma 2.3 for $f \in \mathbf{PR}(\mathbf{N})$ (cf. 2.3) to the case where $f \in \mathbf{R}(\mathbf{N})$, as follows.

- The function $\underline{f}$ for $f \in \mathbf{PR}(\mathbf{N})$ is defined as before.
- If $f \in \mathbf{PR}(\mathbf{N})$, we define $\underline{\mu f} = \mu_{\underline{\mathbf{N}}} \underline{f}$; that is,

$$\underline{\mu f}(\vec{t}) = t \iff t = \underline{\min\{m \in \mathbf{N} | f(\vec{t}, \underline{m}) = \mathsf{nil}\}}.$$

- $\underline{g \circ (g_1, \ldots, g_l)} = \underline{g} \circ (\underline{g_1}, \ldots, \underline{g_l})$.

Then the functions $\underline{f}$ where $f \in \mathbf{R}(\mathbf{N})$ satisfy the following.

1. $\underline{f} \in \mathbf{R}_{\underline{\mathbf{N}}}(\mathbf{T})$.
2. $\underline{f}(\underline{m_1}, \underline{m_2}, \ldots, \underline{m_n}) = \underline{m} \iff f(m_1, m_2, \ldots, m_n) = m$.
   (In particular, $\underline{f}(\underline{m_1}, \ldots, \underline{m_n})$ is defined iff $f(m_1, \ldots, m_n)$ is defined.)

3. $f_\nu = \theta^{-1} \circ \underline{f} \circ \theta$.

Proof By induction on $\mathbf{R(N)}$. (For details, see [3].) □

3.5 Corollary $\mathbf{R(N)}_\nu \subseteq \mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)}$ where $\mathbf{R(N)}_\nu = \{f_\nu | f \in \mathbf{R(N)}\}$.

Proof The inclusion follows immediately from Lemmas 3.4 and 2.5. □

3.6 Lemma $\mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)} \subseteq \{f_P | P \text{ is a while-program }\}$.

Proof By induction on $\mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)}$. For example, the $\underline{\mathbf{N}}$-minimization function $\mu_{\underline{\mathbf{N}}} f$ of $f \in \mathbf{PR(T)}$ can be computed by the while-program

input($\vec{x}$);
y := nil;
while $f(\vec{x}, y) \neq$ nil do [y := cons(nil, y)];
output(y)

□

3.7 Lemma $\{f_P | P \text{ is a while-program }\} \subseteq \mathbf{R(N)}_\nu$; that is, functions computable by while-programs over $\mathbf{T}$ are conjugates via ν of recursive functions over $\mathbf{N}$.

Proof Given a while-program P over $\mathbf{T}$, we construct a while-program Q over $\mathbf{N}$ which simulates computation of P step by step. The program Q is obtained from P by simply replacing each primitive recursive function g ($\in \mathbf{PR(T)}$) in P with its conjugate $g_{\nu^{-1}} = \nu \circ g \circ \nu^{-1}$ ($\in \mathbf{PR(N)}$) and nil with 0. Thus, for assignment statements

$$x := g(\vec{y}) \text{ becomes } x := g_{\nu^{-1}}(\vec{y}),$$

and for termination conditions in while statements

$$g(\vec{y}) \neq \text{nil} \text{ becomes } g_\tau(\vec{y}) \neq 0.$$

Under this construction, we can observe the equivalence

$$f_P(t_1, ..., t_n) = t \iff f_Q(\nu(t_1), ..., \nu(t_n)) = \nu(t)$$

between the function f_P computed by while-program P over $\mathbf{T}$ and the function f_Q computed by while-program Q over $\mathbf{N}$. In other words, we have $f_P = \nu^{-1} \circ f_Q \circ \nu = (f_Q)_\nu$. This completes the proof, since $f_Q \in \mathbf{R(N)}$. □

3.8 Theorem $\mathbf{R(N)}_\nu = \mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)} = \{f_P | P \text{ is a while-program }\}$.

Proof Immediate from Corollary 3.5 and Lemmas 3.6 and 3.7. □

Next, we see the relation between the two classes of recursive functions over $\mathbf{T}$.

3.9 Theorem $\mathbf{R(T)} = \mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)}$.

Proof To see the inclusion $\subseteq$, it suffices to verify that the $\mathbf{T}$-minimization function $\mu_{\mathbf{T}} g$ belongs to $\mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)}$ for each $g \in \mathbf{PR(T)}$, since by definition the class $\mathbf{R}_{\underline{\mathbf{N}}}\mathbf{(T)}$ includes $\mathbf{PR(T)}$ and closed under composition. But it is easy; the following while-program over $\mathbf{T}$ clearly computes the function $\mu_{\mathbf{T}} g$.

input($\vec{x}$);
y := nil;
while $g(\vec{x}, y) \neq$ nil do [y := next(y)];
output(y)

To see $\supseteq$, all we need is to show that for each $g \in \mathbf{PR(T)}$ the $\underline{\mathbf{N}}$-minimization function $\mu_{\underline{\mathbf{N}}} g$ belongs to $\mathbf{R(T)}$. For the purpose, we define a new function

$$g'(\vec{t}, t) = \begin{cases} g(\vec{t}, t) & \text{if } t \in \underline{\mathbf{N}}, \\ g(\vec{t}, \mathsf{nil}) & \text{otherwise,} \end{cases}$$

and note the equivalence

$$(\mu_{\underline{\mathbf{N}}} g)(\vec{t}) = t \iff (\mu_{\mathbf{T}} g')(\vec{t}) = t.$$

Here we have $g' \in \mathbf{R(T)}$ since $g'(\vec{t}, t) = \mathsf{if}(\underline{\mathbf{N}}?(t), g(\vec{t}, t), g(\vec{t}, \mathsf{nil}))$ (cf. Examples 2.2.3). Thus we get $\mu_{\underline{\mathbf{N}}} g = \mu_{\mathbf{T}} g' \in \mathbf{R(T)}$. □

3.10 Corollary $\mathbf{R(N)}_\nu = \mathbf{R(T)}$.

In [3], $\mathbf{R(T)}$ is shown to be equal to the class of tree functions which are representable by (type-free) λ-calculus. Due to space limitation, we omit the details.

4. Choice of Coding Functions

In this section, we study how the choice of coding function $\nu : \mathbf{T} \to \mathbf{N}$ affects the class $\mathbf{PR(N)}_\nu$ of conjugates of primitive recursive numeric functions and the class $\mathbf{R(N)}_\nu$ of conjugates of recursive numeric functions. The proof idea is originally due to [2] Chapter III.

As before, we write f_φ for the conjugate $\varphi^{-1} \circ f \circ \varphi$ of function f via bijection φ, and extend the notation to classes F of functions as $F_\varphi = \{f_\varphi | f \in F\}$.

4.1 Lemma Suppose $a : \mathbf{N} \to \mathbf{N}$ and $b : \mathbf{T} \to \mathbf{T}$ are bijections. Then

1. If $\mathsf{suc}_a \in \mathbf{PR(N)}$, then $a^{-1} \in \mathbf{PR(N)}$,
2. If $\mathsf{cons}_b \in \mathbf{PR(T)}$, then $b^{-1} \in \mathbf{PR(T)}$.

Proof In the second case, the function b^{-1} can be defined by primitive recursion since

$$b^{-1}(\mathsf{cons}(t', t'')) = (b^{-1} \circ \mathsf{cons} \circ b)(b^{-1}(t'), b^{-1}(t'')).$$

where $b^{-1} \circ \mathsf{cons} \circ b = \mathsf{cons}_b \in \mathbf{PR(T)}$. The first case is similar. □

4.2 Lemma Suppose $a : \mathbf{N} \to \mathbf{N}$ and $b : \mathbf{T} \to \mathbf{T}$ are bijections. Then

1. $\mathbf{PR(N)}_a = \mathbf{PR(N)} \iff a, a^{-1} \in \mathbf{PR(N)}$,
2. $\mathbf{PR(T)}_b = \mathbf{PR(T)} \iff b, b^{-1} \in \mathbf{PR(T)}$.

Proof For the direction $\Rightarrow$ of the second case, since the condition implies $\mathsf{cons}_b \in \mathbf{PR(T)}$, we know $b^{-1} \in \mathbf{PR(T)}$ from Lemma 4.1.2. Since the condition can also be stated as $\mathbf{PR(T)}_{b^{-1}} = b \circ \mathbf{PR(T)} \circ b^{-1} = \mathbf{PR(T)}$, we get $b = (b^{-1})^{-1} \in \mathbf{PR(T)}$ by the same argument. For the direction $\Leftarrow$, the assumption $b, b^{-1} \in \mathbf{PR(T)}$ implies

$$b \circ \mathbf{PR(T)} \circ b^{-1} \subseteq \mathbf{PR(T)} \quad \text{and} \quad b^{-1} \circ \mathbf{PR(T)} \circ b \subseteq \mathbf{PR(T)},$$

from which we obtain $\mathbf{PR(T)} \subseteq b^{-1} \circ \mathbf{PR(T)} \circ b \subseteq \mathbf{PR(T)}$. The first case is similar. □

Based on these facts, we show a necessary and sufficient condition for a bijection $\varphi : \mathbf{T} \to \mathbf{N}$ to satisfy $\mathbf{PR(N)}_\varphi = \mathbf{PR(T)}$ (cf. Theorem 2.8).

4.3 Theorem For any bijection $\varphi : \mathbf{T} \to \mathbf{N}$, the following conditions are equivalent.

1. $\mathbf{PR(N)}_\varphi = \mathbf{PR(T)}$.

2. $\mathrm{suc}_\varphi \in \mathbf{PR(T)}$, and $\mathrm{cons}_{\varphi^{-1}} \in \mathbf{PR(N)}$.

Proof The direction $1 \Rightarrow 2$ is obvious. To see the direction $2 \Rightarrow 1$, suppose $\varphi : \mathbf{T} \to \mathbf{N}$ satisfies the condition 2. Then

$$\nu^{-1} \circ \varphi \circ \mathrm{cons} \circ \varphi^{-1} \circ \nu \in \nu^{-1} \circ \mathbf{PR(N)} \circ \nu = \mathbf{PR(T)}$$

because the standard coding function $\nu : \mathbf{T} \to \mathbf{N}$ satisfies 1 by Theorem 2.8. This, together with Lemma 4.1.2, implies that the bijection $\nu^{-1} \circ \varphi$ belongs to $\in \mathbf{PR(T)}$ since $\nu^{-1} \circ \varphi = (\varphi^{-1} \circ \nu)^{-1}$. Similarly, since the condition 2 implies

$$\nu \circ \varphi^{-1} \circ \mathrm{suc} \circ \varphi \circ \nu^{-1} \in \nu \circ \mathbf{PR(T)} \circ \nu^{-1} = \mathbf{PR(N)},$$

we get $\nu \circ \varphi^{-1} \in \mathbf{PR(N)}$ by Lemma 4.1.1. Then

$$\begin{aligned} (\nu^{-1} \circ \varphi)^{-1} &= \varphi^{-1} \circ \nu \\ &= \nu^{-1} \circ (\nu \circ \varphi^{-1}) \circ \nu \\ &\in \nu^{-1} \circ \mathbf{PR(N)} \circ \nu = \mathbf{PR(T)}. \end{aligned}$$

Thus we get $(\nu^{-1} \circ \varphi)$, $(\nu^{-1} \circ \varphi)^{-1} \in \mathbf{PR(T)}$, which then implies

$$\begin{aligned} \mathbf{PR(T)} &= (\nu^{-1} \circ \varphi)^{-1} \circ \mathbf{PR(T)} \circ (\nu^{-1} \circ \varphi) && \text{by Lemma 4.2.2} \\ &= \varphi^{-1} \circ (\nu \circ \mathbf{PR(T)} \circ \nu^{-1}) \circ \varphi \\ &= \varphi^{-1} \circ \mathbf{PR(N)} \circ \varphi && \text{by Theorem 2.8.} \end{aligned}$$

This completes the proof. □

From the theorem, we know that change of the coding function does not affect the notion of conjugates of primitive recursive functions as long as the conjugate of $\mathrm{suc} : \mathbf{N} \to \mathbf{N}$ and that of $\mathrm{cons} : \mathbf{T}^2 \to \mathbf{T}$ remain to be primitive recursive. A similar statement holds in the case of recursive functions, as we see below.

4.4 Lemma Suppose $a : \mathbf{N} \to \mathbf{N}$ and $b : \mathbf{T} \to \mathbf{T}$ are bijections. Then

1. If $\mathrm{suc}_a \in \mathbf{R(N)}$, then $a^{-1} \in \mathbf{R(N)}$,

2. If $\mathrm{cons}_b \in \mathbf{R(T)}$, then $b^{-1} \in \mathbf{R(T)}$.

Proof The proof idea of Lemma 4.1 is applicable to this lemma, because both $\mathbf{R(N)}$ and $\mathbf{R(T)}$ are closed under primitive recursion. □

4.5 Lemma Suppose $a : \mathbf{N} \to \mathbf{N}$ and $b : \mathbf{T} \to \mathbf{T}$ are bijections. Then

1. $\mathbf{R(N)}_a = \mathbf{R(N)} \iff a, a^{-1} \in \mathbf{R(N)}$,

2. $\mathbf{R(T)}_b = \mathbf{R(T)} \iff b, b^{-1} \in \mathbf{R(T)}$.

Proof As Lemma 4.2. □

Based on above lemmas and Corollary 3.10 that shows $\mathbf{R(N)}_\nu = \mathbf{R(T)}$ for the standard coding function ν, we can prove the following counterpart of Theorem 4.3 for recursiveness.

4.6 Theorem For any bijection $\varphi : \mathbf{T} \to \mathbf{N}$, the following conditions are equivalent.

1. $\mathbf{R(N)}_\varphi = \mathbf{R(T)}$.

2. $\mathrm{suc}_\varphi \in \mathbf{R(T)}$, and $\mathrm{cons}_{\varphi^{-1}} \in \mathbf{R(N)}$.

Proof As Theorem 4.3. □

4.7 Corollary For any bijection $\varphi : \mathbf{T} \to \mathbf{N}$, the following conditions are equivalent.

1. $\mathrm{suc}_\varphi \in \mathbf{PR(T)}$, and $\mathrm{cons}_{\varphi^{-1}} \in \mathbf{PR(N)}$.
2. $\mathbf{PR(N)}_\varphi = \mathbf{PR(T)}$.
3. $\mathbf{PR(N)}_\varphi = \mathbf{PR(T)}$, and $\mathbf{R(N)}_\varphi = \mathbf{R(T)}$.

Proof Immediate from Theorems 4.3 and 4.6, and inclusions $\mathbf{PR(N)} \subseteq \mathbf{R(N)}$ and $\mathbf{PR(T)} \subseteq \mathbf{R(T)}$. □

References

[1] W. S. Brainerd and L. H. Landweber (1974). *Theory of Computation*, John Wiley & Sons.

[2] S. Eilenberg and C. C. Elgot (1970). *Recursiveness*, Academic Press.

[3] M. Kimoto (2000). *On Computability of Functions over Binary Trees*, Master Thesis (in Japanese), Department of Mathematical and Computing Sciences, Tokyo Institute of Technology.

[4] D. E. Knuth (1973). *The Art of Computer Programming*, Vol.1 – Fundamental Algorithms, Addison Wesley.

[5] J. McCarthy (1960). Recursive functions of symbolic expressions and their computation by machine, Part I, *Communications of the ACM*, Vol. 3, pp. 184 - 195.

[6] H. E. Rose (1984). *Subrecursion – Functions and Hierarchies*, Clarendon Press, Oxford.

[7] M. Takahashi (1998). A primer on proofs and types, *Theories of Types and Proofs*, MSJ-Memoirs Vol.2 (M.Takahashi, M.Okada, and M.Dezani, eds., Mathematical Society of Japan), pp. 1 - 44.

[8] M. Takahashi (2001). Lambda-representable functions over term algebras, *International Journal of Foundations of Computer Science*, Vol. 12, No. 1, pp. 3 - 29.

[9] J.V.Tucker and J.I.Zucker (2000). Computable functions and semicomputable sets on many-sorted algebras, *Handbook of Logic in Computer Science*, Vol.5 (S.Abramsky, D.M.Gabbay and T.S.E.Maibaum, eds., Oxford University Press), pp. 317 - 525.